Figure 1

```
module EXAMPLE:

input RESET, START; output GOT;

signal REQUEST, GRANT in
  loop abort                          % RESET restarts the loop
    await START;
    emit REQUEST;
    present GRANT then emit GOT end
  ||                                  % run concurrently
    loop
      present REQUEST then emit GRANT end;
      pause;                          % wait for the next cycle
      pause
    end
  when RESET end
end.
```

Figure 2A

p ; q	p; q;

Figure 2B

emit S	S = 1;

Figure 2C

loop p end	for (;;) p;

Figure 2D

present S then p else q end	if (S) p; else q;

Figure 3A

```
pause                    state = k;
                         if (level < 1) level = 1;
                         goto Join;
                    case k:
```

Figure 3B

```
await S                  goto Entry;
                    case k:
                         if (!S) {
                         Entry:
                              state = k;
                              if (level < 1) level = 1;
                              goto Join;
                         }
```

Figure 3C

```
                         goto Entry;
                    case k:
abort                    if (!S)
                              switch (state) {
     body                          Entry: body;
when S                        }
```

Figure 3D

```
                         goto Entry;
                    case k:
suspend                  if (S) {
                              if (level < 1) level = 1;
                              goto Join;
                         }
                         switch (state) {
     body                      Entry: body;
when S                   }
```

Figure 3E

```
signal S in                          S = 0;
                                     goto Entry;
                                case k:
                                     S = 0;
                                     switch (state) {
        body                                 Entry: body;
end                                  }
```

Figure 3F

```
exit T;                              if (level < 2) level = 2;
                                     goto Join;
```

Figure 3G

```
trap T in                          innerLevel = 0;
                                   fork StartA, StartB;
                                 case k:
                                   innerLevel = 0;
                                   fork ResumeA, ResumeB;

                                 ResumeA:
                                   switch (statep) {
      bodyA                        StartA: bodyA;
                                   case 0: ;
                                   }
                                   goto InnerJoin;
||
                                 ResumeB:
                                   switch (stateq) {
      bodyA                        StartB: bodyB;
                                   case 0: ;
                                   }
                                   goto InnerJoin;

                                 InnerJoin:
                                   join;
handle T do                        switch (innerLevel) {
                                   case 1:      /* paused */
                                     state = k;
                                     if (level < 1) level = 1;
                                     goto OuterJoin;
                                   case 2:      /* exited */
      handler                        handler;
end                                  break;
                                   }
```

Figure 4

```
                                Start: goto L0;
                                Resume:
                                switch (s & 0x3) {
pause;                          L0:     s=1; goto Join;
pause;                          case 1: s=2; goto Join;
                                case 2: goto L1;
abort                           case 3: if (!B)
                                        switch (s>>2 & 0x7 ) {
        pause;                          L1:     s=3 | 0<<2; goto Join;
        pause;                          case 0: s=3 | 1<<2; goto Join;
                                        case 1: goto L2;
        abort                           case 2: if (!A)
                                                switch (s>>5) {
                pause;                          L2:     s=3 | 2<<2 | 0<<5; goto Join;
                pause                           case 0: s=3 | 2<<2 | 1<<5; goto Join;
                                                case 1:
                                                }
        when A;                                         s=3 | 2<<3; goto Join;
        pause;                                  case 3: s=3 | 2<<4; goto Join;
        pause                                   case 4:
                                                }
when B                          s = 0; goto Join;
                                case 0: ; /* not running */
                                }
                                Join:
```

Figure 5

```
loop
        trap T in
                loop
                        present A
                        then
                                emit B
                        end;
                        pause;
                end
        ||
                pause;
                exit T;
        end
end
```

```
if (inLaterCycles) {

        if (A) then B = 1;
        /* pause (level 1) */
||
        /* exit T (level 2) */

}

        if (A) then B = 1;
        /* pause (level 1) */
||
        /* pause (level 1) */

inLaterCycles = 1;
```

Figure 6A

```
1   /* THE THREE MAIN DATA TYPES OF ACCFG: CNODE, PROCESS, and THREAD.
2   MAIN DATA TYPE OF SCFG IS SNODE.
3
4   cnode = node in the acyclic concurrent control-flow graph (accfg)
5   snode = node in the sequential control-flow graph (scfg) */
6
7   /* The properties of a cnode are defined as follows: */
8
9   cnode::pthreads;    /* Threads to which this cnode belongs ("parents") Most
10  nodes belong to exactly one thread. The exceptions are join nodes, which belong to each
11  thread they join, and the topmost process, which belongs to no thread. */
12
13  /* A "predecessor" is a (snode, condition) pair that will be used as the source and label
14  respectively of an added arc. Each predecessor is an snode that could run a cnode */
15
16  cnode::runningPredecessors; /* set of normal snodes */
17  cnode::restartPredecessor;  /* restart snode */
18
19  /* The distinction between the two types of predecessor (i.e., "running" and "restart") is
20  used in the "suspend any running thread in process p" routine, which avoids creating
21  save state nodes for restart nodes. */
22
23  cnode::index;   /* integer index of the node. (topological order number) */
24
25  cnode::state;   /* Possible states are: Running, Runnable, or Suspended. Only a
26  "process" can be in a "Running" state, which means it contains a thread which is
27  actively executing. */
```

Figure 6B

```
1  /* A Process is a cnode (and therefore inherits the properties of a cnode) that
2  corresponds to a fork node and contains one or more threads.
3          A process's state may be Suspended, Runnable, or Running.
4          A Suspended process is contained in a thread that is not running.
5          A Runnable process is contained in a thread that is running, but none of the
6  threads contained in the process are running.  A Runnable process is ready to restart one
7  of the threads it contains.
8          A Running process means one of its contained threads is currently running (i.e.,
9  executing instructions).
10
11         Suspending the running thread within a process changes the process's state from
12 Running to Runnable.  This is typically followed by starting (or restarting) another
13 thread, contained within the process, which changes the process's state from Runnable
14 back to Running.  This suspension of one thread and the starting (or restarting) of
15 another thread is also known as a "context switch."
16
17         The properties of a process and a thread are as follows.  */
18
19 process::threads;  /* The threads contained in the process */
20
21 process::runningThread;  /* Indicates which, if any, of the threads contained in
22 the process is the currently running thread. */
23
24 thread::process;  /* Which process contains this thread */
25
26 thread::cnodes;  /* The cnodes in this thread that could be executed next */
27
28 thread::stateVariable;  /* State variable used for saving the state of the thread
29 when the thread is suspended.  This state variable is subsequently read when the thread
30 is resumed. */
```

Figure 6C

```
1   /* MAIN ROUTINE: "synthesize a scfg"
2   This main routine synthesizes the scfg from the input accfg */
3
4   synthesize a scfg
5   {
6   /* INITIALIZATION: Create the outermost process and a single thread within in. Put the
7   first scheduled node in this thread. The thread starts out suspended; the first iteration of
8   the main loop will resume it. */
9
10  en = create the SCFG entry node;
11
12  op = create the outermost process;
13
14  op.state = Runnable;
15
16  op.runningThread = none;
17
18  op.runningPredecessors += (en, -);   /* Entry node "en" is made to be
19  the runningPredecessor of "op" and the edge from op to en has no label as indicated by
20  the hyphen "-". */
21
22  op.pthreads = empty /* By definition, the outermost process is not in a thread. */
23
24  op.restartPredecessor = empty;
25
26  tt = new thread;
27
28  op.threads += tt;
29
30  tt.process = op;
31
32  fn = first node in the schedule;
33
34  /* Set the state variable used by the outermost thread */
35  tt.stateVariable = fn.index
36
37  tt.cnodes += fn;
38
39  fn.pthreads += tt; /* Put the first node in the top thread */
40
41  fn.state = Suspended;
```

Figure 6D

```
1   /* MAIN LOOP: successively assigns to current node "cn" each cnode of the input accfg
2   in order of the topological sort. */
3
4   for each node cn in scheduled order {
5
6         sn = copy node cn and its expression into the SCFG;
7         th = first thread in cn.pthreads; /* Thread of this node */
8
9         /* Rest of this loop is divided into four main code blocks labeled A, B, C and D.
10        For each cnode assigned to cn, a code block from A or B, and a code block
11        selected from C or D, is executed.
12
13        The pair of code blocks selected for execution depends on the type of the cnode,
14        and is illustrated by the following table:
15
16        cnode type:          Normal Fork Join
17        selection from A or B:  B    B    A
18        selection from C or D:  D    C    D
19
20        if ( cn is a join node ) {
21           /* CODE BLOCK A */
22              /* Earlier, this join node would have been placed in all of the threads it
23              was joining.  Run it in its parent's thread. */
24              p = th.process;
25              th = thread in p.pthreads; /* unique since this is a process */
26              switch to thread th;
27              suspend any running thread in p;
28              run cnode p as snode sn;
29              th.cnodes -= p;   /* Delete the now-terminated process */
30
31        } else { /* cn is a Normal or Fork node */
32              /* CODE BLOCK B */
33              switch to thread th;
34              run cnode cn as snode sn;
35              /* We've run cn, so it no longer plays a role in the thread */
36              th.cnodes -= cn;
37        }
```

Figure 6E

```
1        if ( cn is a fork node ) {
2             /* CODE BLOCK C */
3             process = new process;
4             process.state = Runnable;
5             process.runningThread = none;
6             process.runningPredecessors += (sn, -); /* Note that
7             edge from "process" to sn has a empty label */
8             process.restartPredecessor = empty;
9             th.cnodes += process; /* Put the new process in its thread */
10            for ( each successor cns of cn ) {
11                 /* Create a new thread for each successor and put the successor
12                 node in the new thread. */
13                 thread = new thread;
14                 process.threads += thread;
15                 thread.stateVariable = cns.index; /* Set the state
16                 variable for "thread" to have a default value being the topological
17                 index of cns. */
18                 thread.cnodes += cns;
19                 put cnode cns in thread thread;
20                 /* Initialize state of successor */
21                 cns.state = Suspended;
22            }
23
24       } else { /* This is a Normal or Join node */
25            /* CODE BLOCK D */
26            for ( each successor cns of cn ) {
27                 th.cnodes += cns;
28                 put cnode cns in thread th;
29                 cns.runningPredecessors += (sn, edge
30                 condition from cn to cns in the input accfg);
31            } /* end "for ( each successor cns of cn )" */
32       } /* end "else" */
33
34   } /* end MAIN LOOP */
35
36   } /* end "synthesize a scfg" */
```

Figure 6F

```
1   run cnode cn as snode sn
2   {
3   for ( each node snp in cn.runningPredecessors )
4        add an edge from snp to sn, labeled like the
5        predecessor edge from cn to snp;
6
7   if ( cn.restartPredecessor is not empty )
8        add an edge from cn.restartPredecessor to sn, labeled
9        like the predecessor edge from cn to
10       cn.restartPredecessor;
11
12  /* having used these predecessor edges, they should now be removed */
13  cn.runningPredecessor = empty;
14  cn.restartPredecessor = empty;
15  }
16
17
18  put cnode cns in thread th
19  {
20    if th is not already in cns.pthreads,
21      cns.pthreads += th;
22  }
```

Figure 6G

```
1   switch to thread th
2   {
3   /* "switch to thread th" does nothing if the thread is already running.  If the thread is not
4   running, it saves the state of any already-running thread (suspends it) and restarts the
5   desired thread. */
6
7   /* If there is at least one thread above "th," make sure it is also running */
8   if ( th.process.pthreads is not empty )
9         /* The parent thread is unique for a process */
10        switch to thread th.process.pthreads;
11
12  p = th.process;
13
14  /* If a different thread is running, suspend it */
15  if ( p.state == Running AND p.runningThread != th )
16        suspend any running thread in p;
17
18  if ( p.state == Runnable ) {
19        /* Restart our thread by adding a restart node and making this restart node a
20        predecessor of each suspended node. */
21
22        rn = new restart node( th.stateVariable );   /* Build a
23        restart node (of SCFG) which tests state of the stateVariable for thread which is
24        to be switched to.  This stateVariable needs to have been set appropriately when
25        thread th was previously suspended. */
26
27        run cnode p as snode rn;
28
29        for ( each cnode cn in th.cnodes ) {
30              cn.restartPredecessor = (rn, cn.index); /* Create an
31              edge from cn to rn whose label has the value cn.index */
32
33              cn.state = Runnable;
34        }
35
36        p.state = Running;
37        p.runningThread = th;
38
39  } /* end "if (p.state == Runnable)" */
40
41  } /* end "switch to thread th" */
```

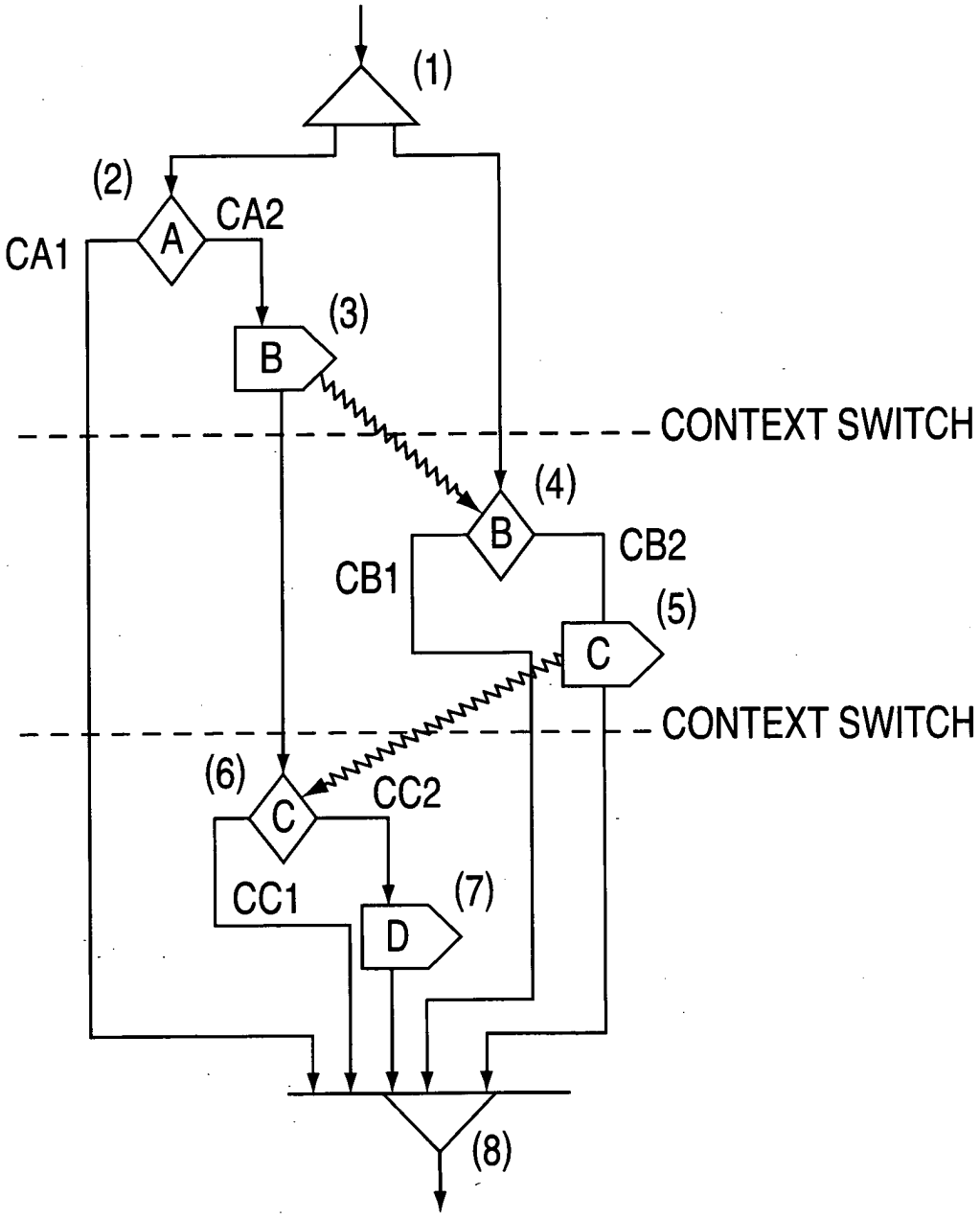
Figure 6H

```
1   suspend any running thread in process p
2   {
3   if ( p.state == Running ) {
4        /* This process has a running thread -- suspend it */
5        p.state = Runnable;
6        th = p.runningThread;
7        restartNode = none; /* Set when the restart node needs a default arc
8        leading from it to suspend this thread */
9
10       /* Save state if there is more than one running cnode in the thread */
11       needToSaveState = true if there is more than one cnode
12       in th;
13       needToSaveState = false if there is not more than one
14       cnode in th;
15
16       /* Suspend each cnode in the the thread */
17
18       for ( each cnode cn in th.cnodes ) {
19
20            /* Suspend any running threads in a process node */
21            if ( cn is a process )
22                 suspend any running thread in cn;
23
24            /* Suspend all running predecessors for this node */
25            if ( cn.runningPredecessors is not empty ) {
26
27                 if ( needToSaveState ) {
28                      sn = new save state node (state for this
29                      thread = cn.index ); /* Makes the "expression"
30                      of sn be the following assignment statement:
31                      th.stateVariable = cn.index. */
32
33                      for ( each snode snp in
34                      cn.runningPredecessors )
35                           add an edge from snp to sn, labeled
36                           like the predecessor edge from cn
37                           to snp;
38
39                      cn.runningPredecessors = empty; /* having
40                      used these predecessor edges, they should now be removed
41                      */
42
43                      p.runningPredecessors += (sn, -); /* add
44                      an edge from p.runningPredecessors to sn, with no label */
```

Figure 6I

```
1                    } else { /* do not save state */
2                         for ( each snode snp in
3                         cn.runningPredecessors )
4                              p.runningPredecessors += (snp, take
5                              label from the edge cn to snp);
6
7                         cn.runningPredecessors = empty; /* having
8                         used these predecessor edges, they should now be removed
9                         */
10                   } /* end "else" */
11
12              } /* end "if ( cn.runningPredecessors is not empty )" */
13
14              /* Rembmer the restart node if this node has a restart predecessor. */
15
16              if ( cn.restartPredecessor is not empty ) {
17                restartNode = cn.restartPredecessor;
18                /* Remove this precessor edge since it is empty */
19                cn.restartPredecessor = empty;
20              }
21
22              cn.state = Suspended;
23
24         } /* end "for ( each cnode cn in th.cnodes )" */
25
26       p.runningThread = none;
27
28       if ( restartNode is not none ) {
29         /* At least one node had a restart predecessor: make sure an arc with a default
30         condition is added from the restart node to handle this condition */
31           p.runningPredecessors += (restartNode, -);
32       }
33
34  } /* end if (p.state == Running) */
35
36  } /* end "suspend any running thread in process p" */
```

"M&A For Converting A CCFG Into A SCFG," to S. Edwards, App. No. 09/477,688.



INPUT ACCF-G

FIG. 7

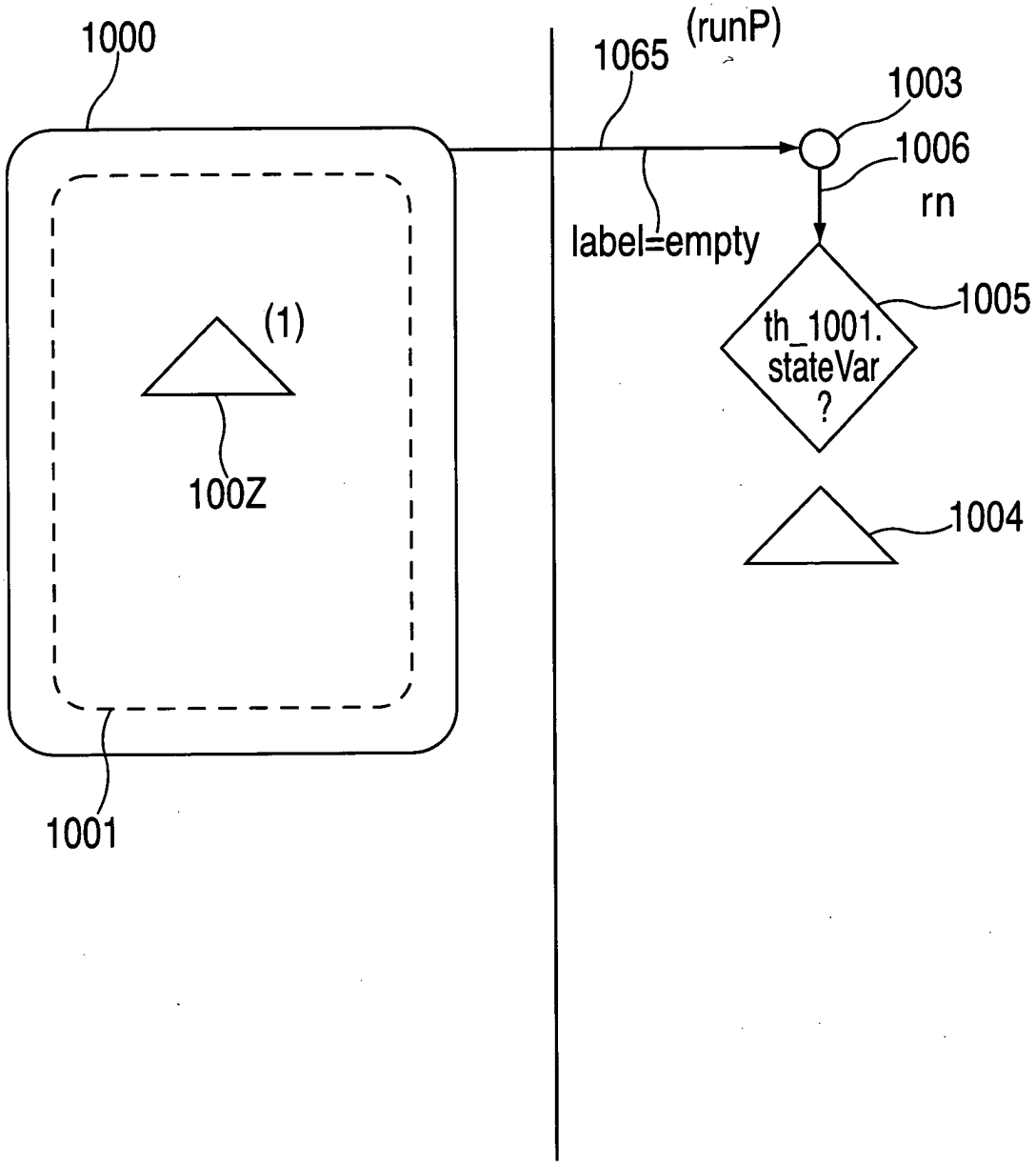

1000
1065
(runP)
1003
1006
rn
label=empty
1005
th_1001.
stateVar
?
(1)
100Z
1004
1001

FIG. 8A

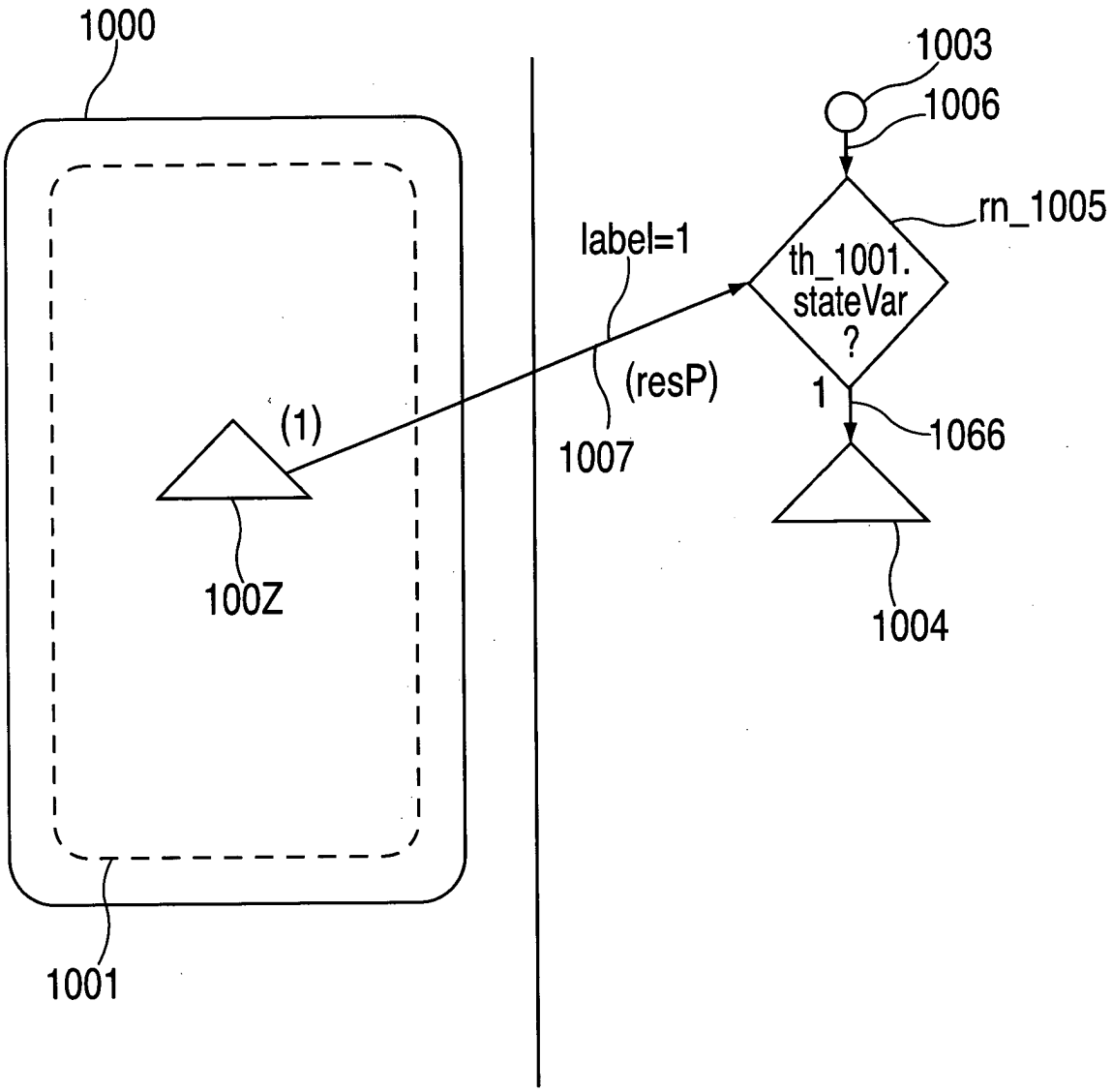

1000
1003
1006
rn_1005
label=1
th_1001.
stateVar
?
(resP)
1
1066
(1)
1007
100Z
1004
1001

FIG. 8B

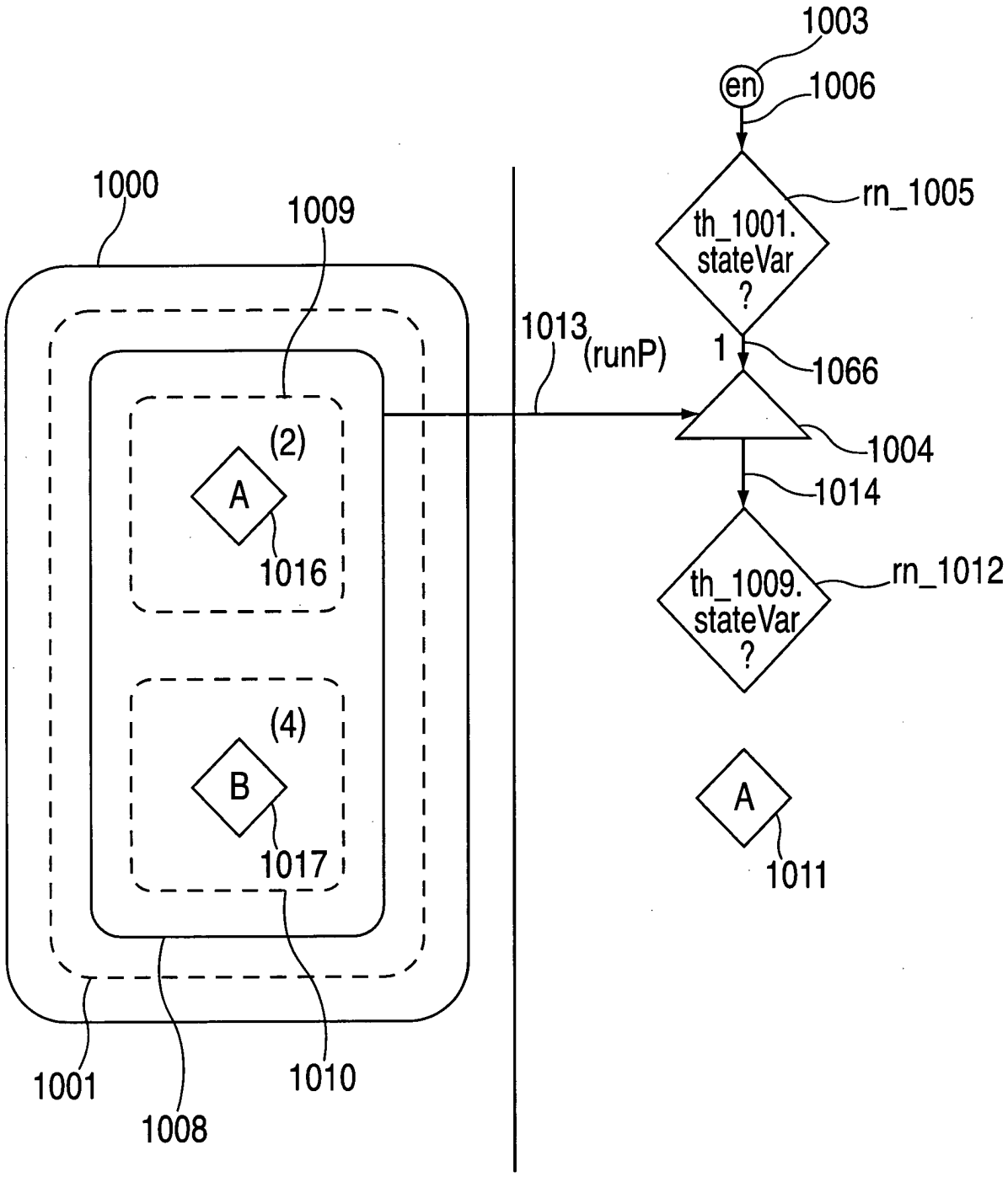

1003
en
1006
rn_1005
th_1001.
stateVar
?
1
1066
1013
(runP)
1004
1014
th_1009.
stateVar
?
rn_1012
A
1011
1000
1009
(2)
A
1016
(4)
B
1017
1001
1008
1010

FIG. 8C

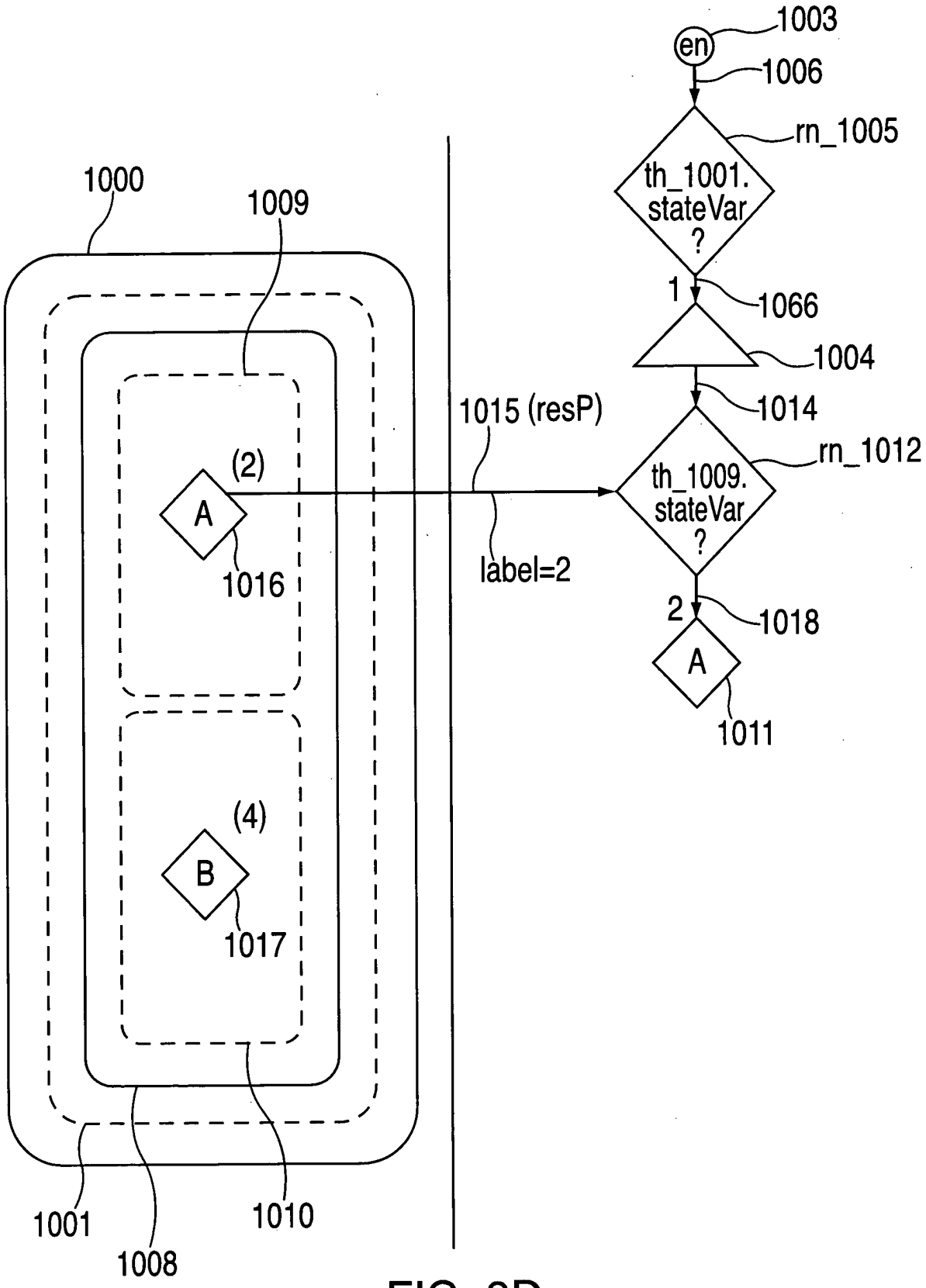

1003
en
1006
rn_1005
th_1001.
stateVar
?
1
1066
1004
1014
rn_1012
th_1009.
stateVar
?
2
1018
A
1011
1000
1009
1015 (resP)
(2)
A
1016
label=2
(4)
B
1017
1001
1008
1010

FIG. 8D

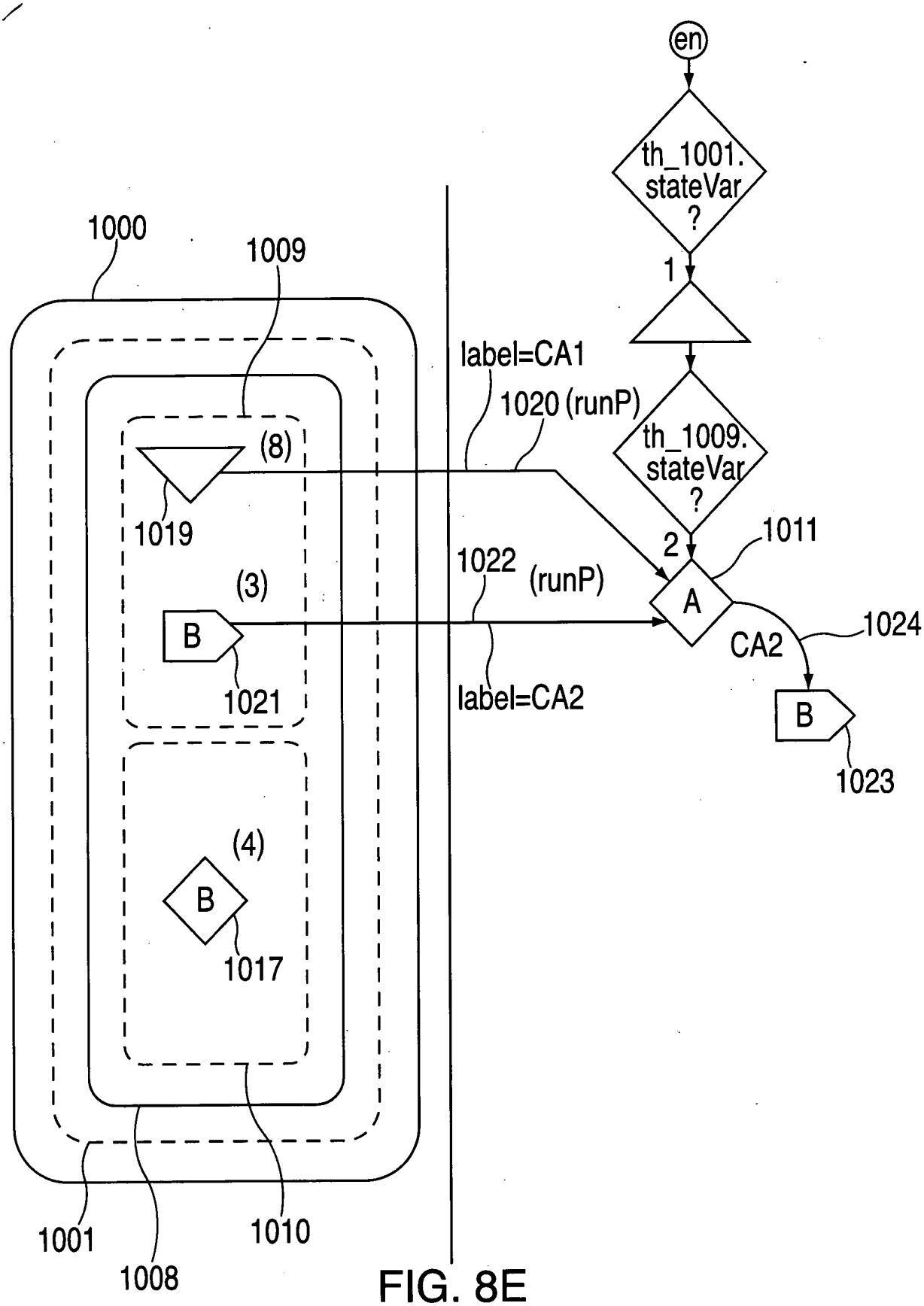

en
th_1001.
stateVar
?
1
th_1009.
stateVar
?
2
1000
1009
label=CA1
1020 (runP)
(8)
1019
1011
1022
(runP)
(3)
B
1021
A
1024
CA2
B
1023
label=CA2
(4)
B
1017
1001
1008
1010

FIG. 8E

"M&A For Converting A CCFG Into A SCFG," to S. Edwards, App. No. 09/477,688.

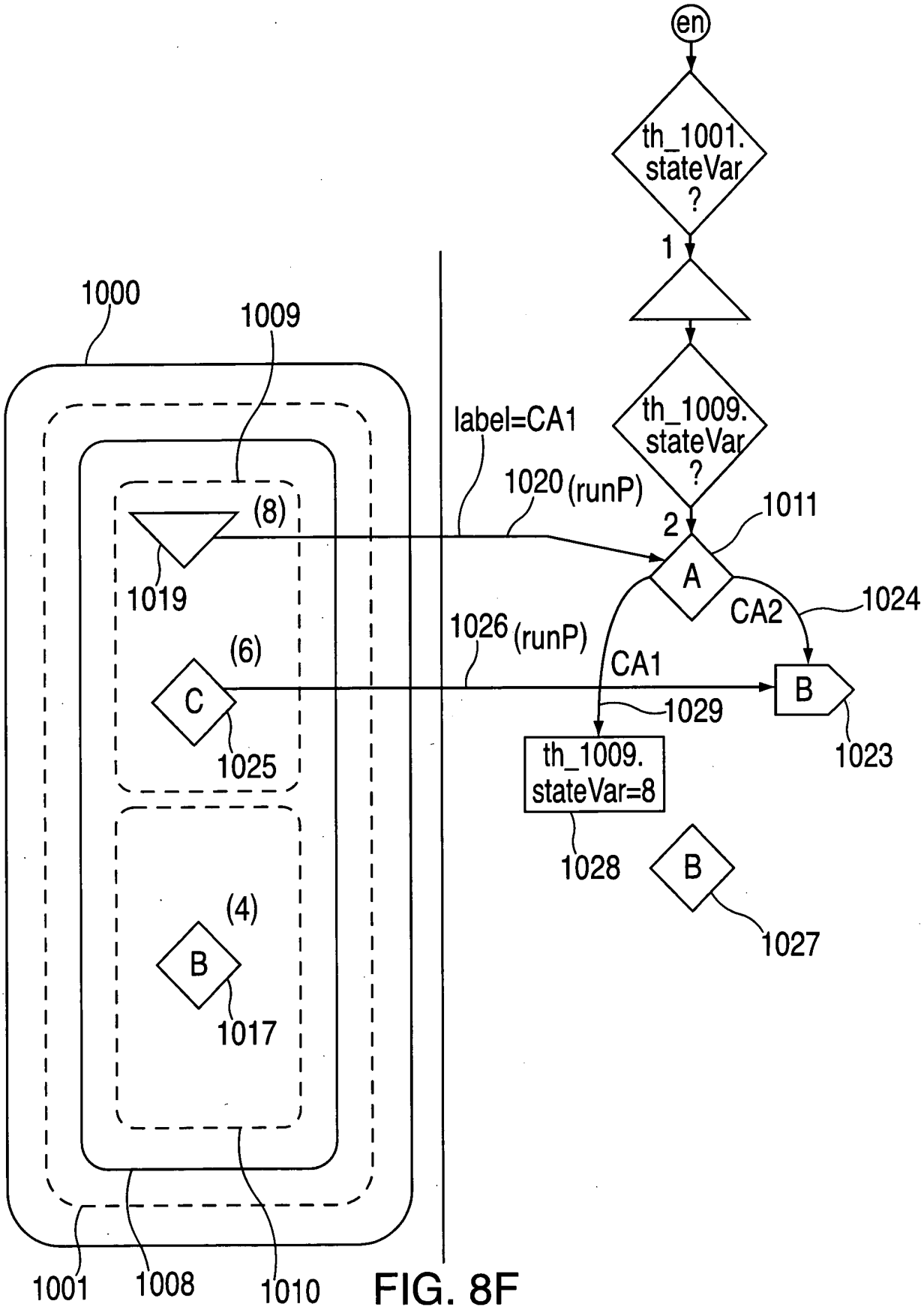


FIG. 8F

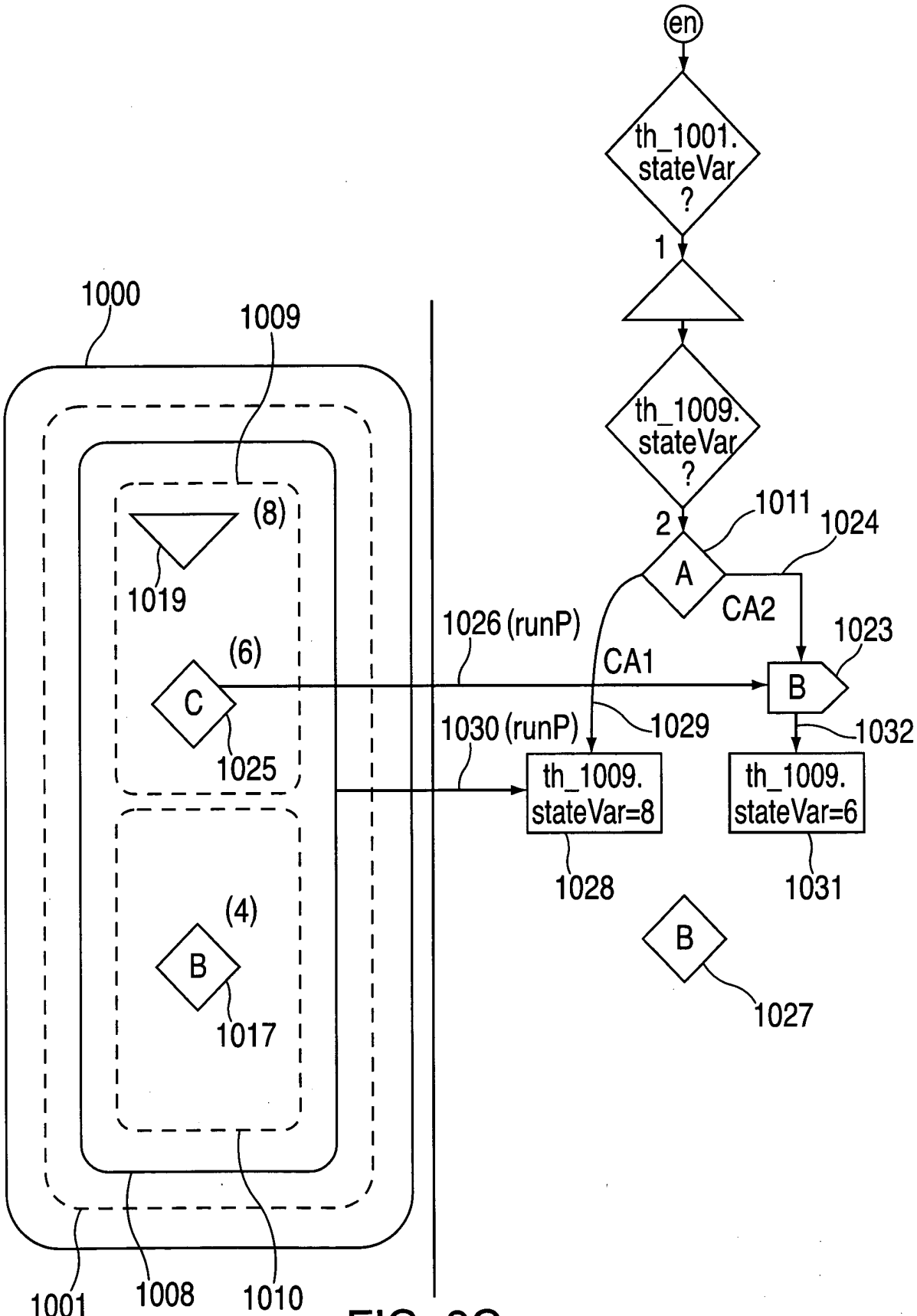

en
th_1001.
stateVar
?
1
th_1009.
stateVar
?
2
1011
1024
A
CA2
CA1
1023
B
1026 (runP)
1030 (runP)
1029
1032
th_1009.
stateVar=8
th_1009.
stateVar=6
1028
1031
B
1027
1000
1009
(8)
1019
(6)
C
1025
(4)
B
1017
1001
1008
1010

FIG. 8G

"M&A For Converting A CCFG Into A SCFG," to S. Edwards, App. No. 09/477,688.

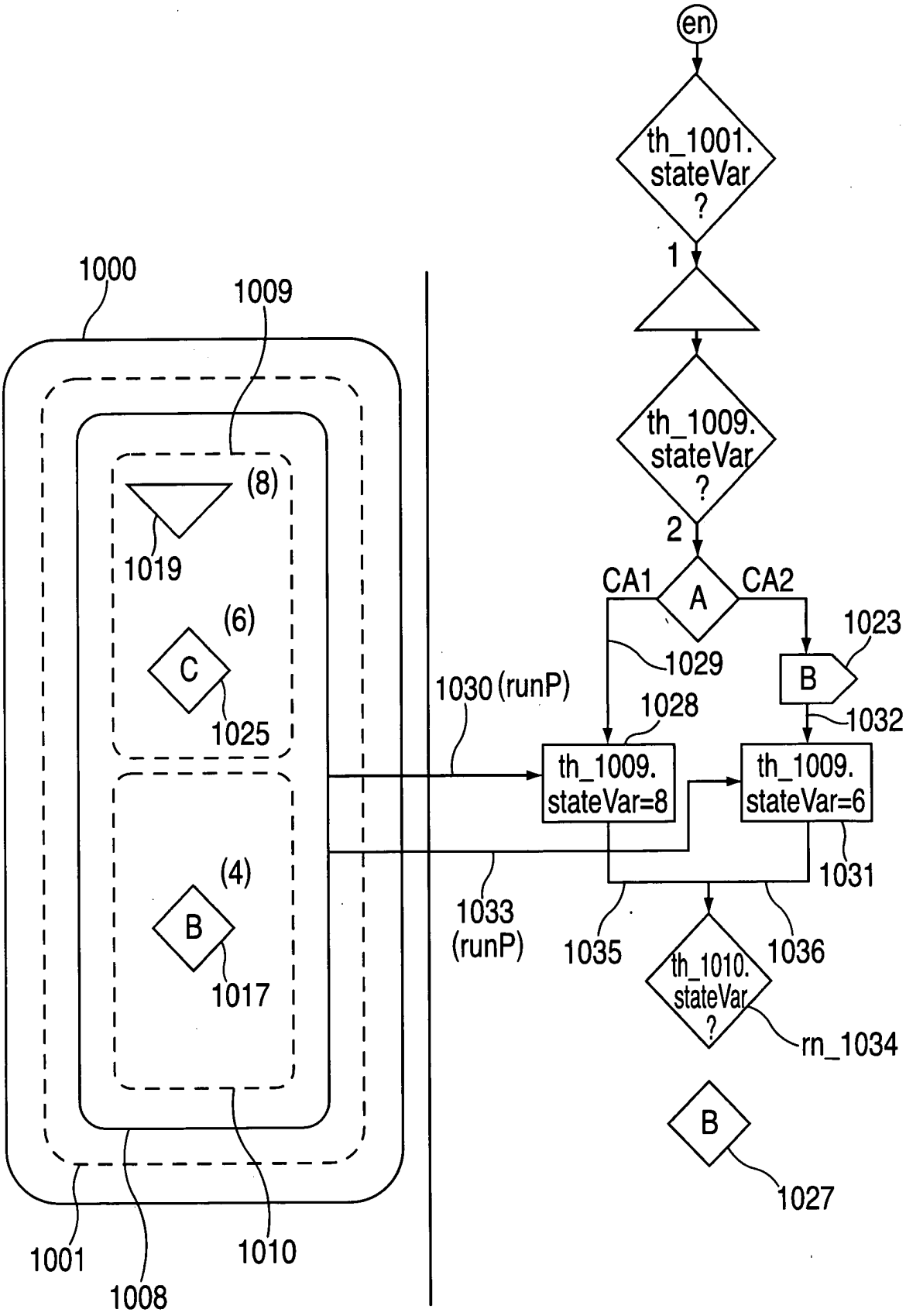


FIG. 8H

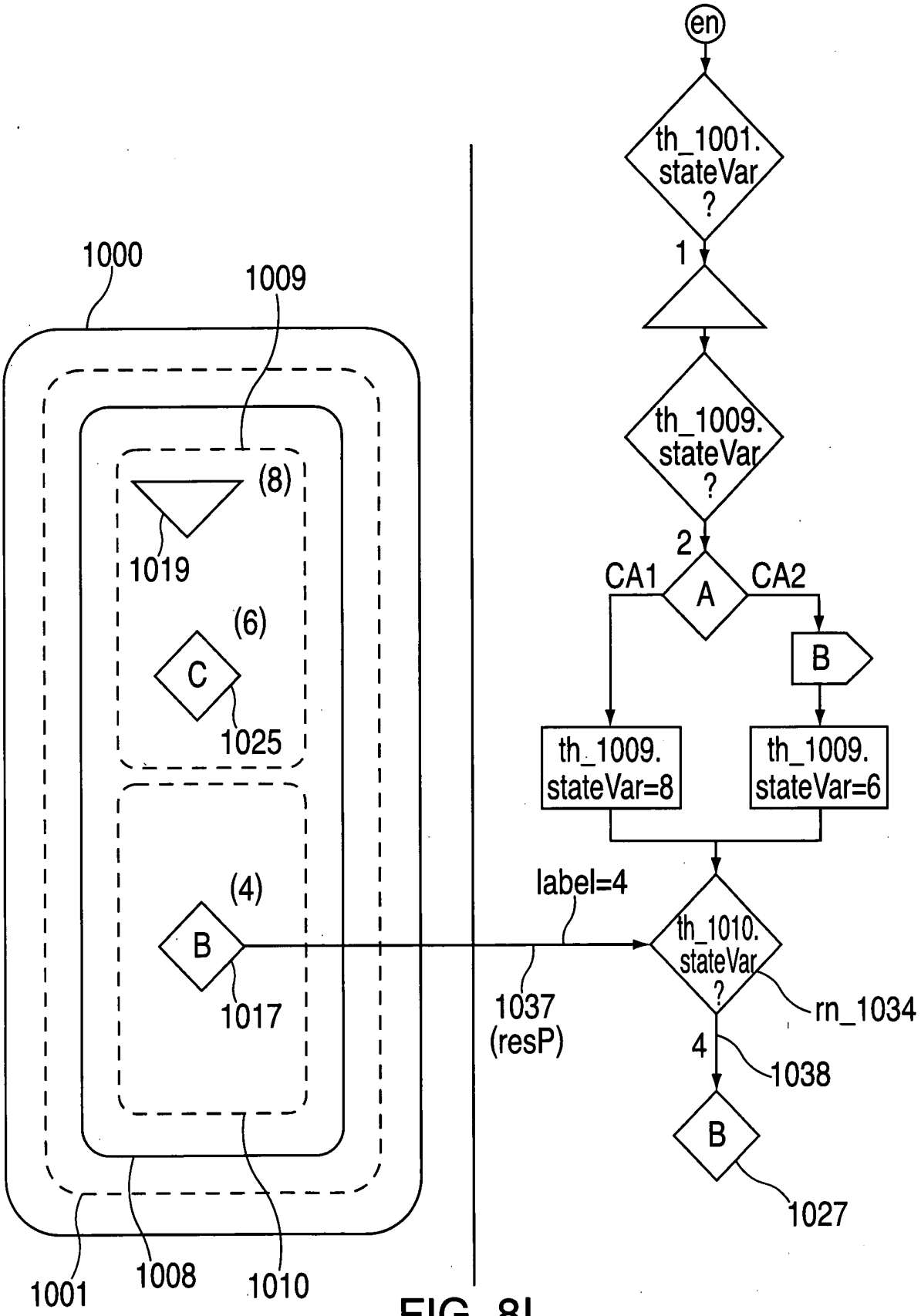

en
th_1001.
stateVar
?
1
th_1009.
stateVar
?
2
CA1
A
CA2
B
th_1009.
stateVar=8
th_1009.
stateVar=6
label=4
th_1010.
stateVar
?
rn_1034
4
1038
B
1027
1037
(resP)
1000
1009
(8)
1019
(6)
C
1025
(4)
B
1017
1001
1008
1010

FIG. 8I

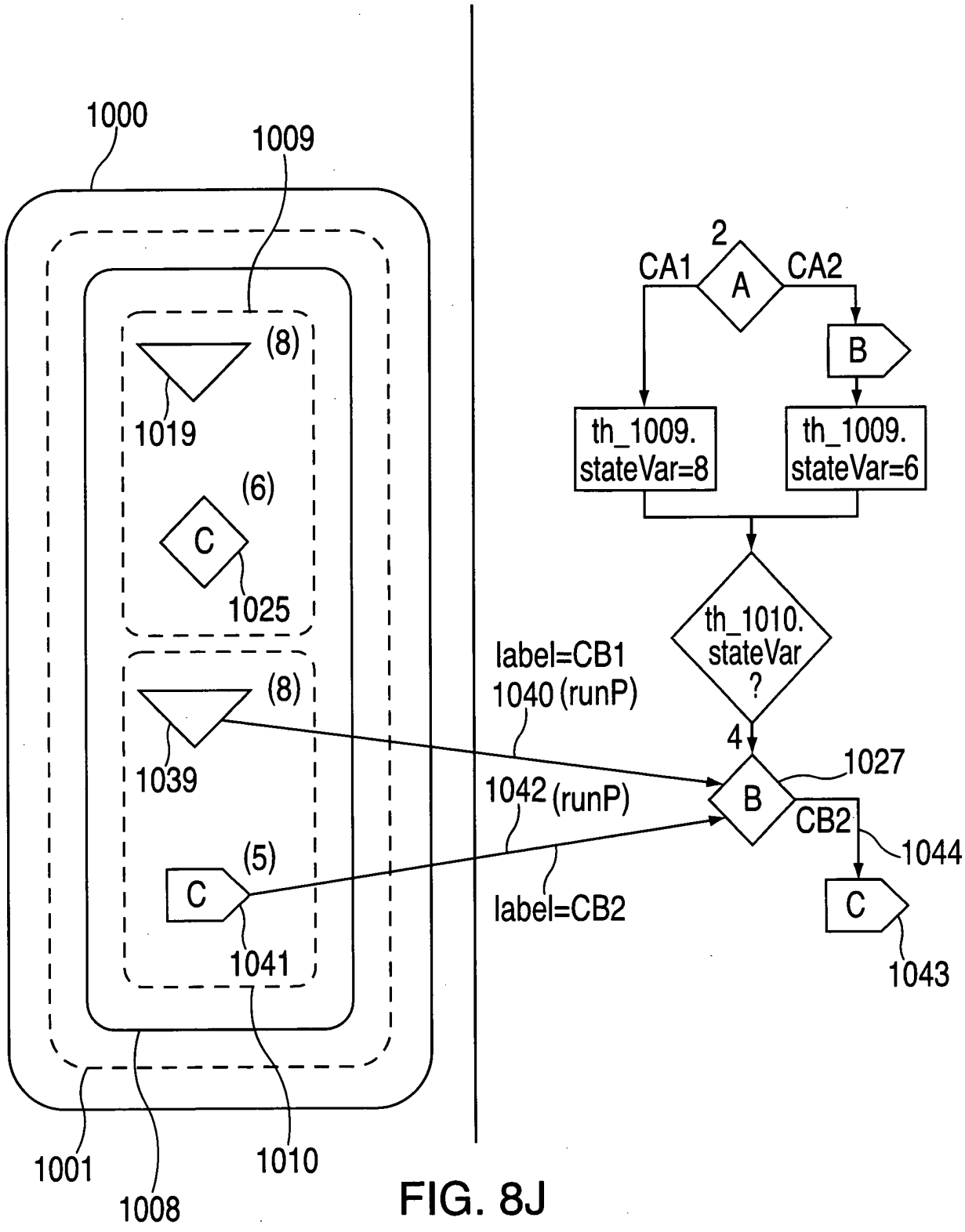

1000
1009
(8)
1019
(6)
C
1025
(8)
1039
(5)
C
1041
1001
1008
1010
2
CA1
A
CA2
B
th_1009.
stateVar=8
th_1009.
stateVar=6
th_1010.
stateVar
?
4
label=CB1
1040 (runP)
1042 (runP)
label=CB2
1027
B
CB2
1044
C
1043

FIG. 8J

"M&A For Converting A CCFG Into A SCFG," to S. Edwards, App. No. 09/477,688.

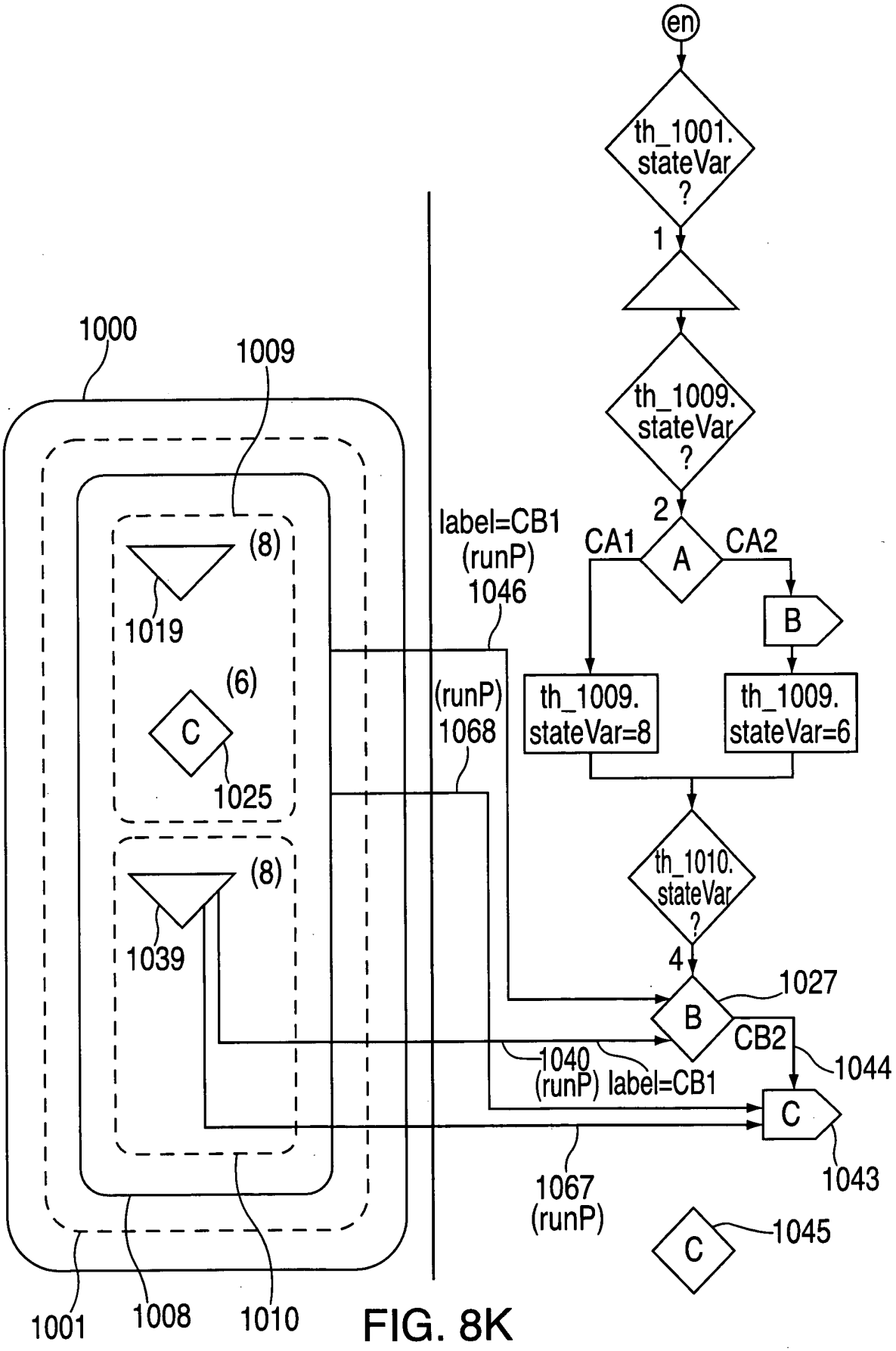


FIG. 8K

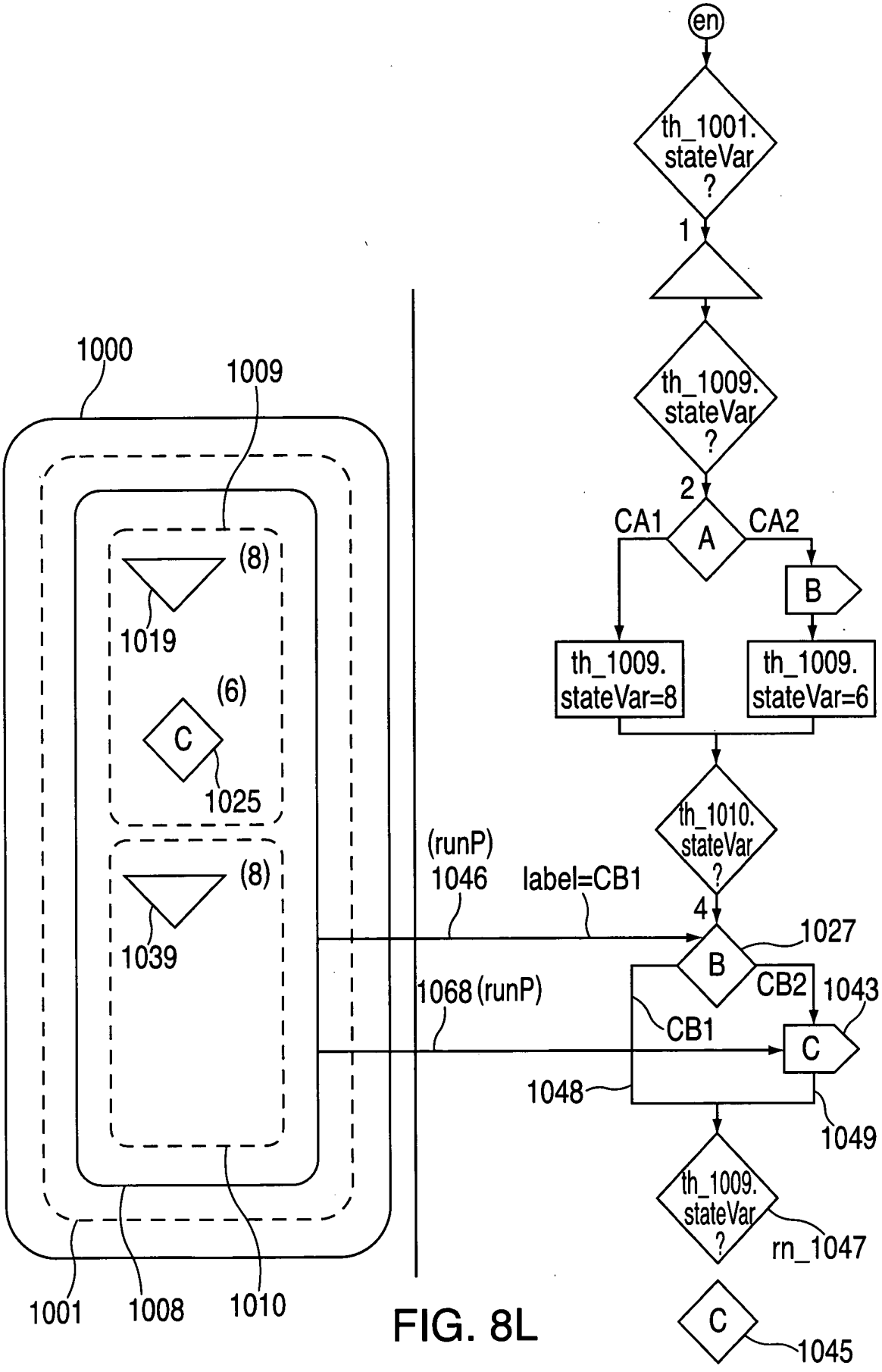

en
th_1001.
stateVar
?
1
th_1009.
stateVar
?
2
CA1
A
CA2
B
th_1009.
stateVar=8
th_1009.
stateVar=6
th_1010.
stateVar
?
4
1000
1009
(8)
1019
(6)
C
1025
(8)
1039
(runP)
1046
label=CB1
1027
B
CB2
1043
1068 (runP)
CB1
C
1048
1049
th_1009.
stateVar
?
rn_1047
C
1045
1001
1008
1010

FIG. 8L

"M&A For Converting A CCFG Into A SCFG," to S. Edwards, App. No. 09/477,688.

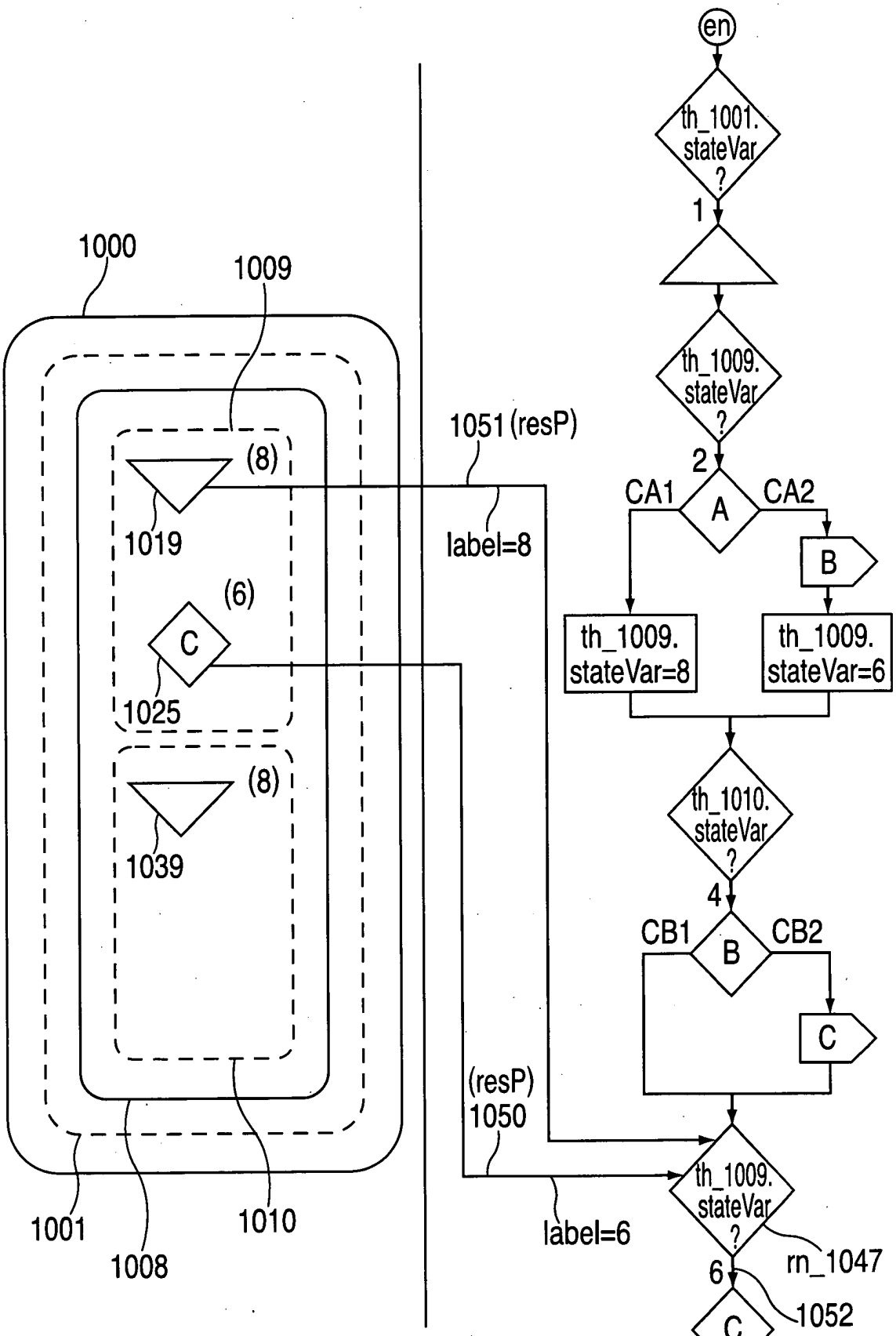


FIG. 8M

"M&A For Converting A CCFG Into A SCFG," to S. Edwards, App. No. 09/477,688.

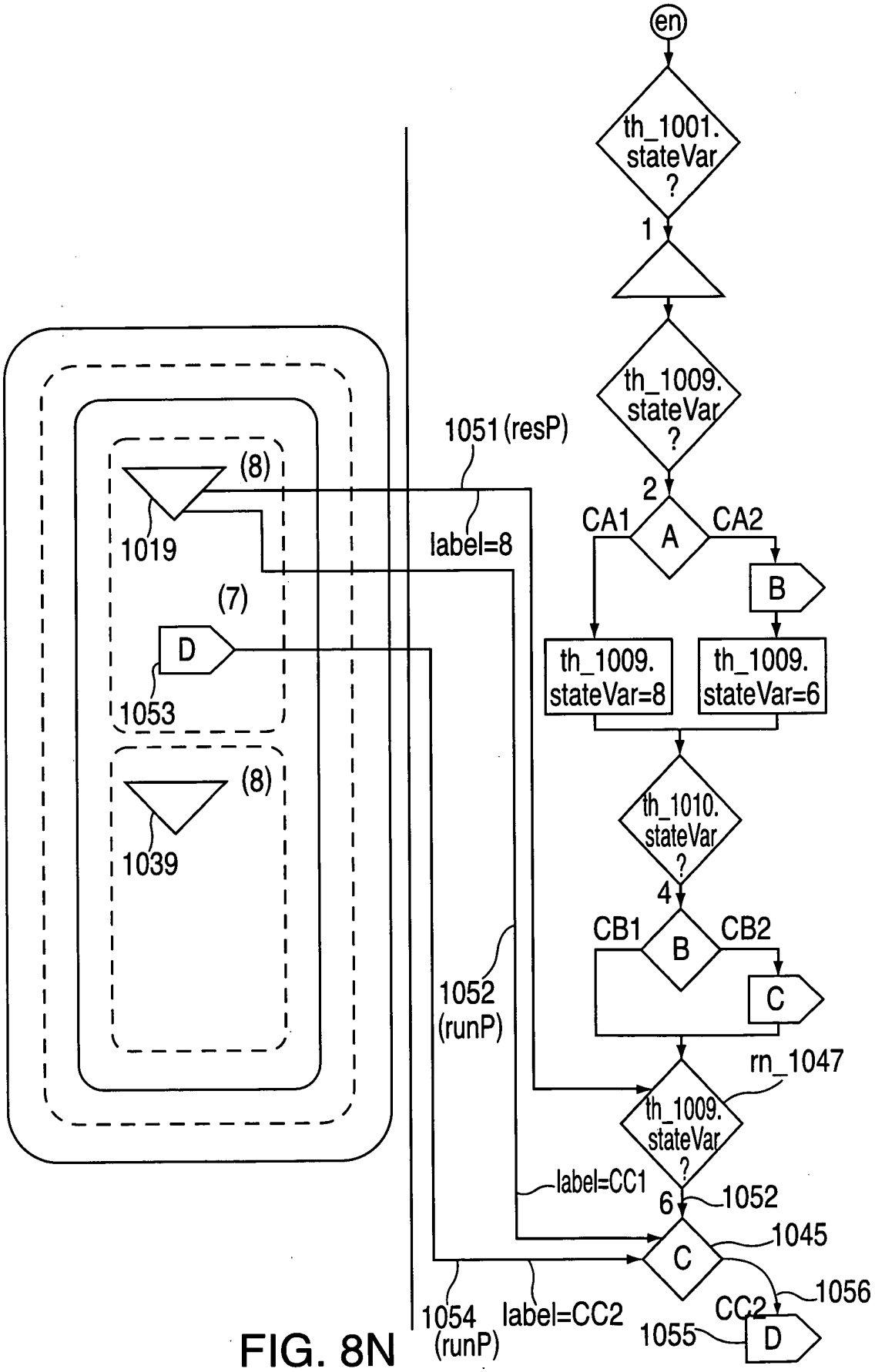


FIG. 8N

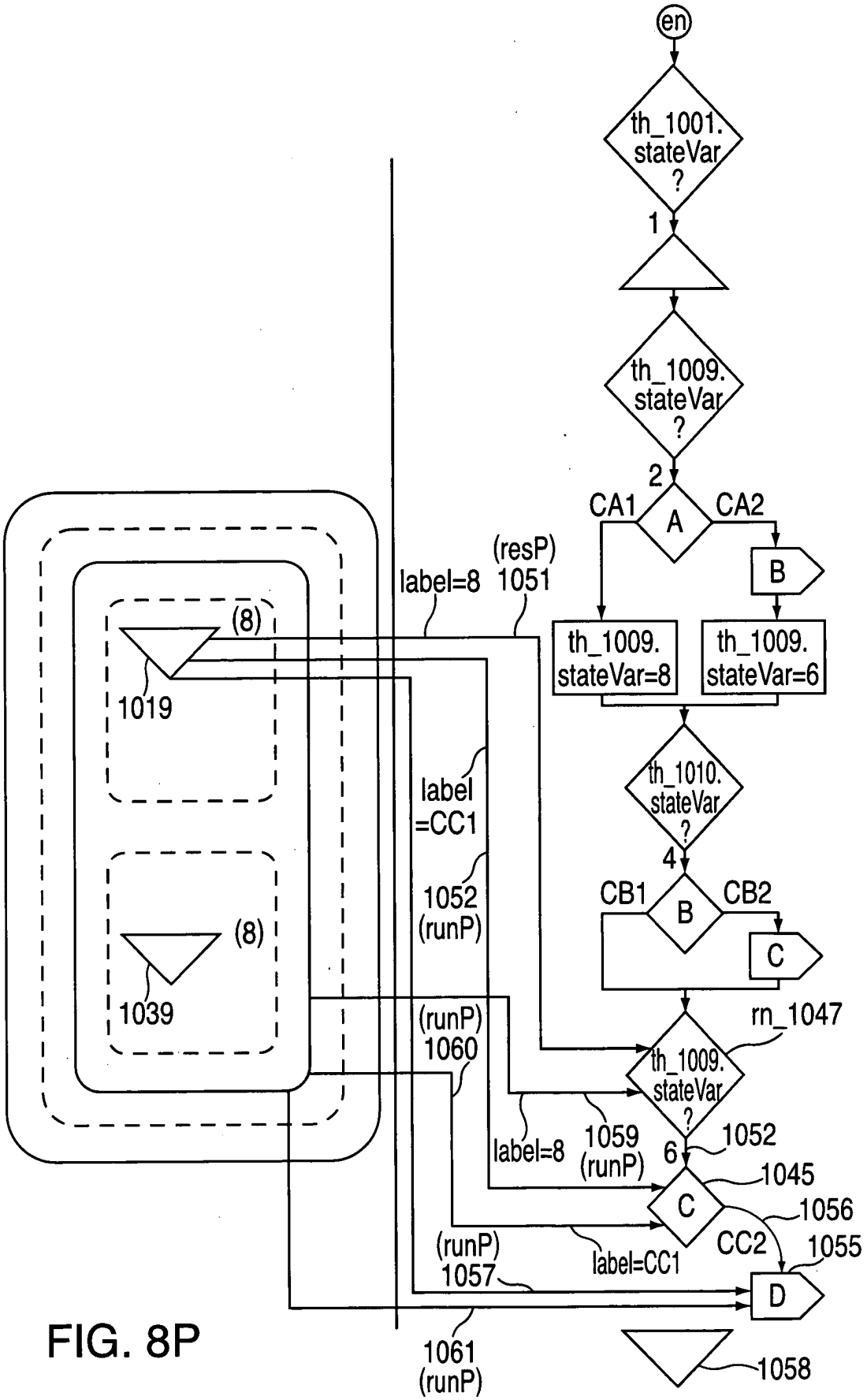

en
th_1001.
stateVar
?
1
th_1009.
stateVar
?
2
CA1
A
CA2
B
th_1009.
stateVar=8
th_1009.
stateVar=6
th_1010.
stateVar
?
4
CB1
B
CB2
C
rn_1047
th_1009.
stateVar
?
6
1052
1045
C
1056
CC2
1055
D
1058
(resP)
1051
label=8
(8)
1019
(8)
1039
label
=CC1
1052
(runP)
(runP)
1060
label=8
1059
(runP)
(runP)
1057
label=CC1
1061
(runP)

FIG. 8P

"M&A For Converting A CCFG Into A SCFG," to S. Edwards, App. No. 09/477,688.

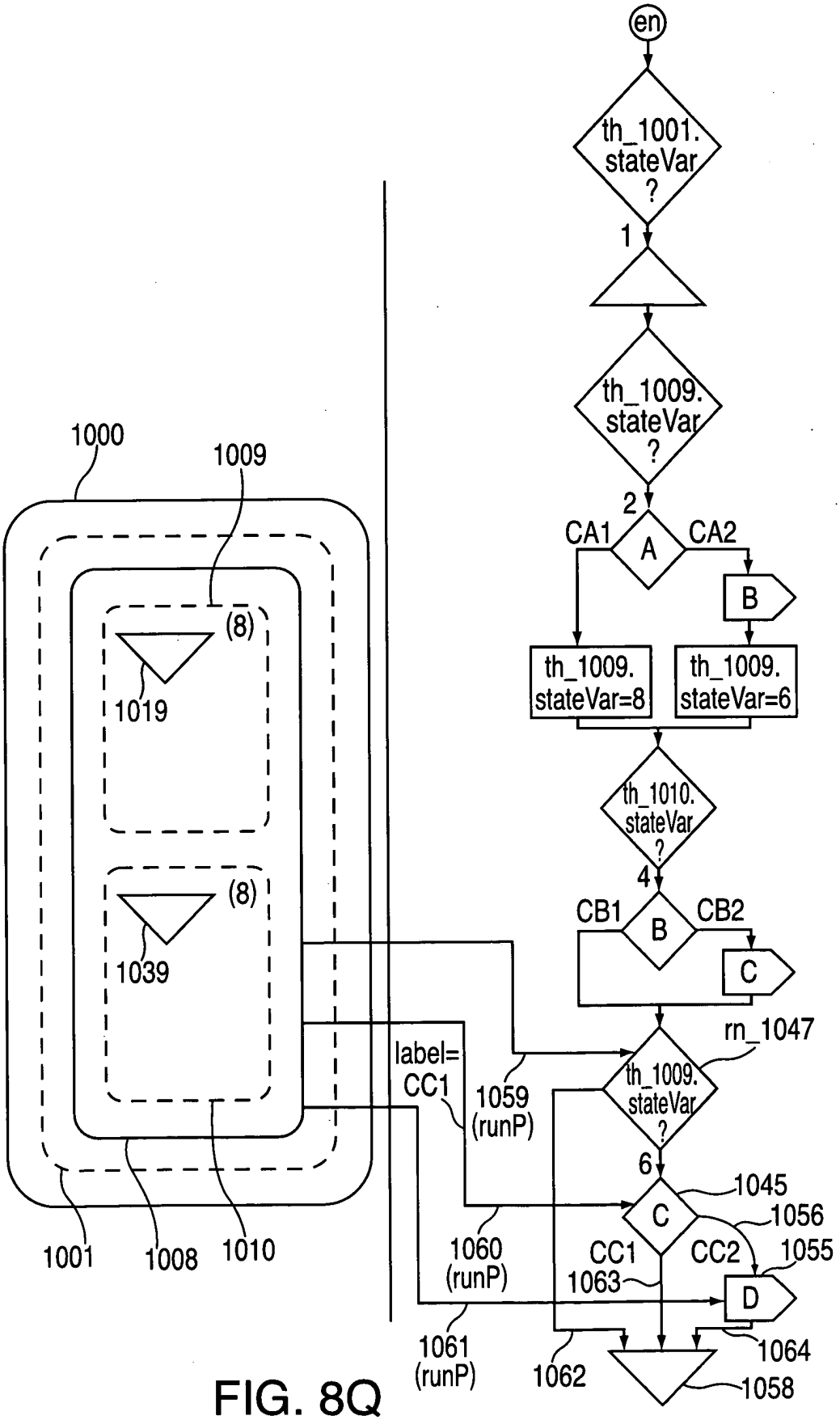


FIG. 8Q

"M&A For Converting A CCFG Into A SCFG," to S. Edwards, App. No. 09/477,688.

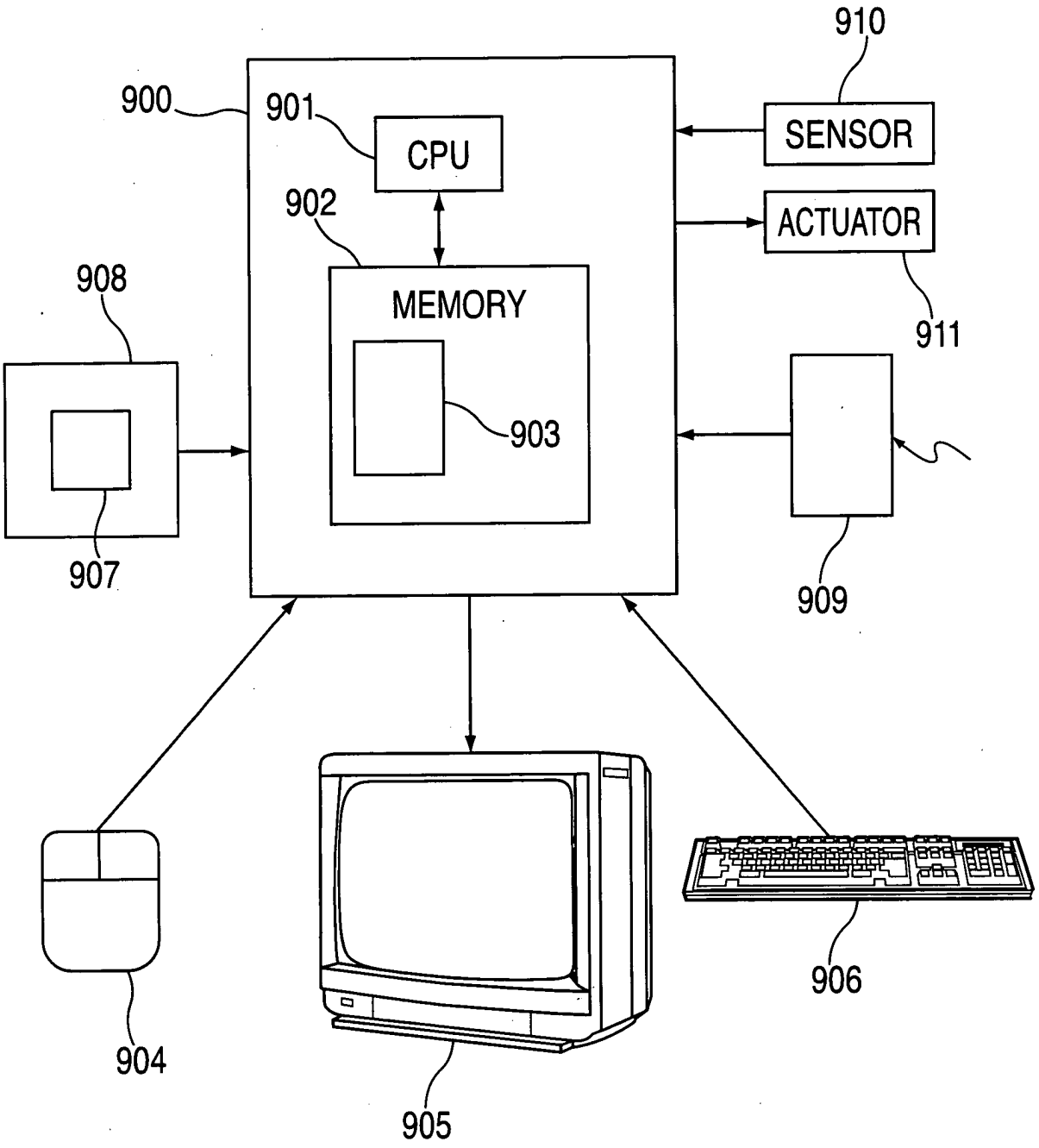


FIG. 9

"M&A For Converting A CCFG Into A SCFG," to S. Edwards, App. No. 09/477,688.

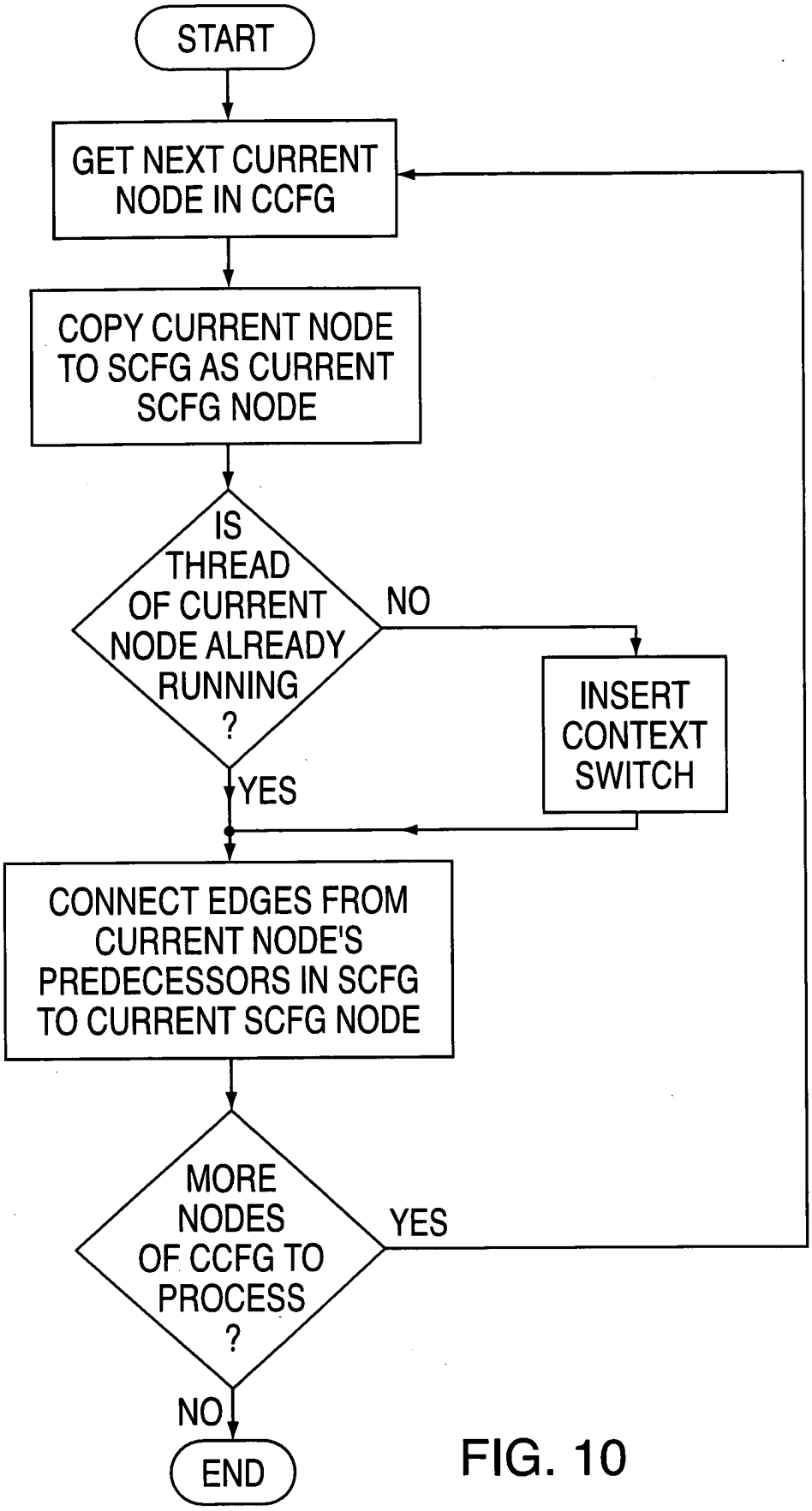


FIG. 10